Technical Document 1670
April 1990

Linear Algebra On a CRAY X-MP

R. F. Freund

NAVAL OCEAN SYSTEMS CENTER

San Diego, California 92152-5000

J. D. FONTANA, CAPT, USN
Commander

R. M. HILLYER
Technical Director

ADMINISTRATIVE INFORMATION

This document was produced by Code 423 of the Naval Ocean Systems Center.

Released by
R. E. Pierson, Head
Ashore Command Centers
Branch

Under authority of
J. A. Salzmann, Head
Ashore Command and
Intelligence Centers
Division

JJ

1.0 VECTORIZATION. We will use the CRAY X-MP for analyzing certain aspects of linear algebra computation on vector supercomputers. Most of the other scientific vector computers are quite similar in architectural design. The main results apply to a wide variety of vector architectures.

1.1. X-MP ARCHITECTURE. While there are many important special features of a vector supercomputer such as a CRAY, undoubtedly the most important in making it "super" is its vectorization capablility. Here we will examine some of the hardware features of vectorization. Later we will look at performance assessment.

1.2 X-MP CONCURRENCY. Vectorization is one of three levels of concurrency (vectorization or parallelism) on a CRAY X-MP [1988]. All three are shown in figure 1.1.

1.2.1. SEPARATE PROCESSORS. First there is the parallelism that comes from having four separate, independent processors or CPUs (the "4" in X-MP 48 refers to four CPUs, the "8" to the number of megawords of main memory). Although hardware and software exist to coordinate these CPUs and make them work together on the same task, most CRAY sites use them as four separate computing machines. Thus, a typical user, using one CPU, sees an average, sustained performance of between 25 and 30 MFLOPS (Million Floating-Point Operations Per Second). This means the average throughput from an X-MP with four processors is around 110 MFLOPS.

1.2.2. SCALAR AND VECTOR SECTIONS. A second level of parallelism resides within each processor, which has both a scalar section (further divided into data and address portions) and a vector section. Each section has its own set of registers for holding operands. Each section also has its own set of functional units to carry out the non floating-point instuctions of shift, integer add, population count, and logical operations. The units for floating point add, floating point multiply, and floating point divide are shared by both the vector and scalar sections. The scalar and vector sections can also be independently programmed in assembly language. However, the most common way to use them (as the CRAY FORTRAN compiler does) is to complement each other on the same task, with the scalar section often preparing data for the vector section or doing those calculations that cannot be vectorized.

1.2.3. VECTORIZATION. The third level of concurrency is vectorization itself, a concept that stems historically from the earlier one of pipelining. Most computational hardware operations take more than one clock cycle. Segmentation is the hardware division of an operation into distinct, one-cycle substages that can be carried out individually. Pipelining is a method of using more than one of these segments at once on different operands. If an operation, say a floating-point add, taking six cycles, is segmented into six separate and independent physical substages, then it is possible to have six adds going on at once, each one at one of the six stages of completion. With no pipelining there would be only one add operation every six cycles. In complete pipelining every stage of the computational pipeline would be used with different operands. In the case of the floating-point add, with six stages, we obtain a sixfold increase in performance, since (after a start-up period) one result is produced every cycle. In a scalar machine, practical difficulties generally permit only partial pipelining, i.e., not all stages are used at once. Vectorization is a simple way to carry out complete pipelining, assuming a long enough data vector. On the CRAY this is done through the use of vector registers, which store a number of operands, as opposed to scalar registers, which each hold only one. Each CPU of the CRAY has eight vector registers, each with the capacity for 64 operands. There are also special vector mask and vector length registers, as well as access ports to memory and to the scalar portion of the CPU.

1.3. EXAMPLE. Let us begin our examination of the mechanics of vectorization by looking at a simple example (figure 1.2). Suppose we have two one-dimensional arrays (vectors), A and B, containing floating-point numbers. Suppose further we wish to add the first 64 values of A to the first 64 values of B and to store the result in another vector, C. Whether using a scalar or vector machine, we could write the FORTRAN code to do this as.

```
      DO 100 I = 1, 64
      C(I) = A(I) + B(I)
100   CONTINUE
```

What happens in vector mode is the compiler directs vector registers, say V0 and V1, to be loaded with the first 64 elements of A and B respectively. After a short delay, the values of V0 and V1 are pipelined through the floating-point-add functional unit. The first add takes another six cycles, after which the first result, C(1), is fed into another register, say V2, and the succeeding 63 C values are pipelined into V2 at a rate of one every cycle. The V2 values are then stored in the memory for the vector C.

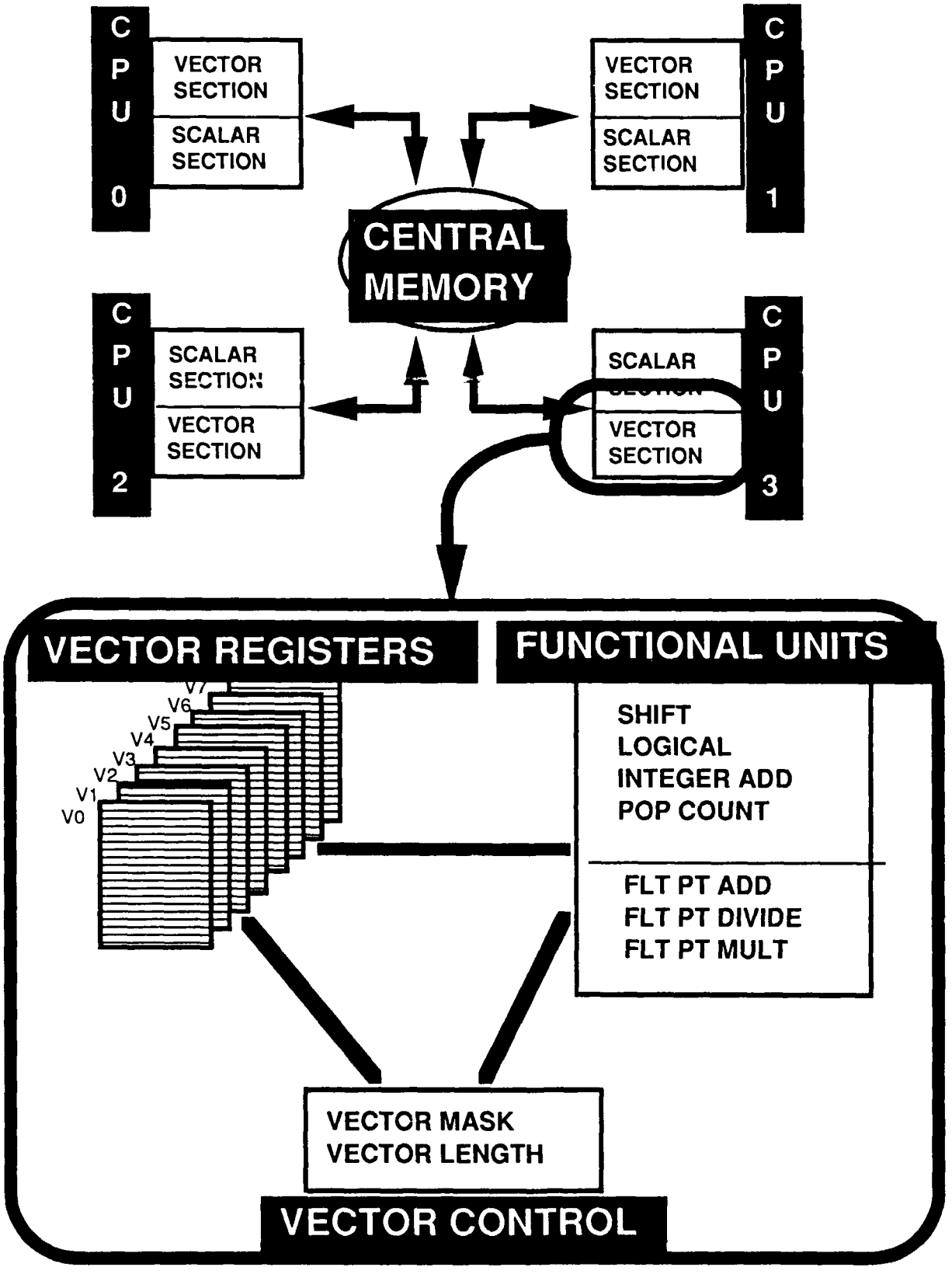


Figure 1.1. *CRAY X-MP, 4 processor configuration.*

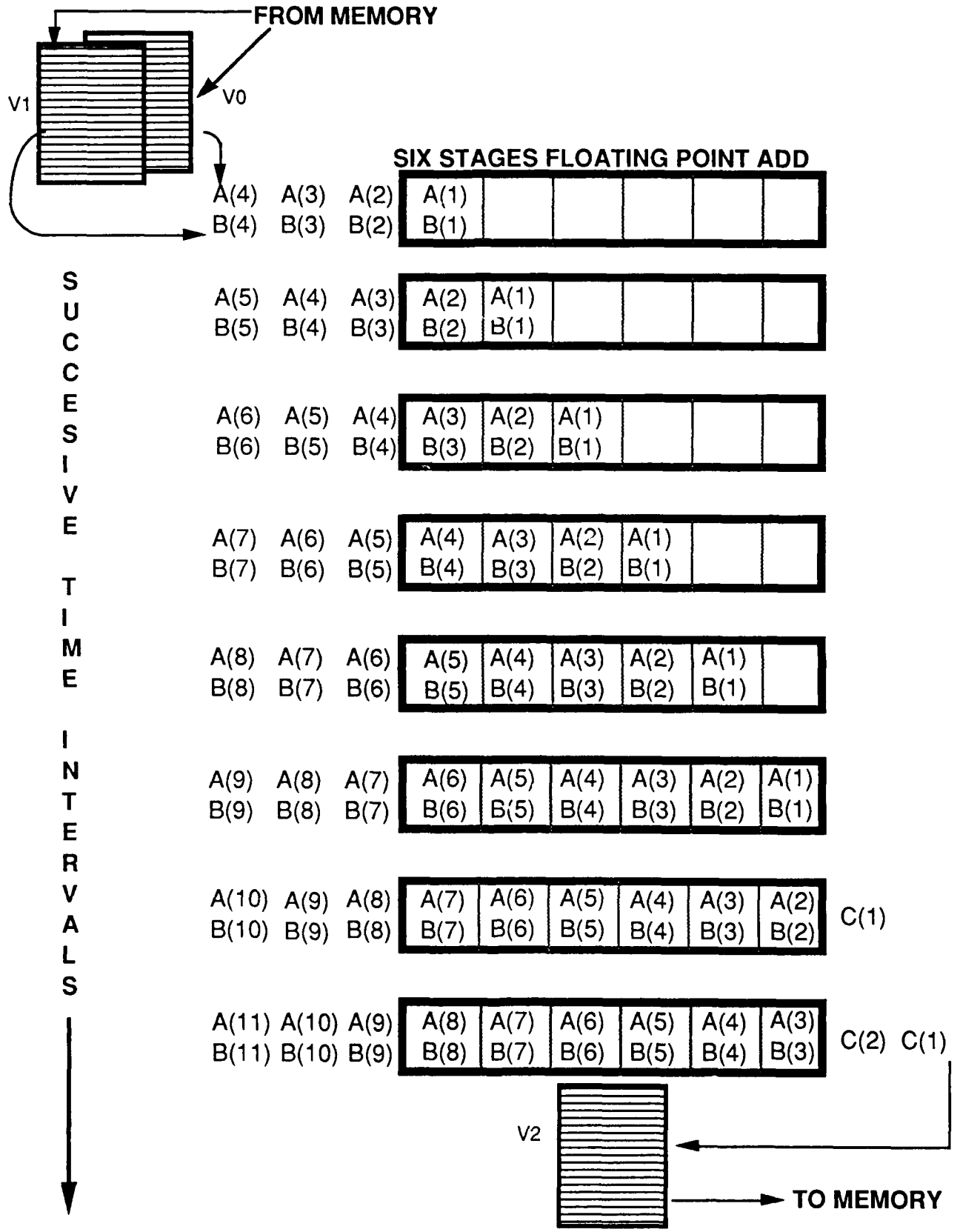


Figure 1.2. *Vectorization of C(I) = A(I) + B(I).*

1.4. VECTOR OPERATIONS. Vector operations can also use scalar operands. For example, if we wished to multiply the first 64 elements of a vector A by a scalar B, we would write the FORTRAN as

```
      DO 100 I = 1, 64
      C(I) = A(I) * B
100   CONTINUE
```

In this case a vector register is loaded with the first 64 elements of A, and a scalar register with B. The A values are pipelined through the multiply unit using B as the other multiplicand. The results, C, are fed into some other vector register and then stored in memory.

Vector operations can be longer than 64, indeed as long as the memory limitations of a particular code will permit. In this case a technique known as "strip-mining" is used, i.e., portions of the vectors are taken sequentially. For example consider the case

```
      DO 100 I = 1, 150
      C(I) = A(I) + B(I)
100   CONTINUE
```

The compiler would generate code to do the first 22 values as one vector operation, the next 64 values, and finally the last 64 values. Thus if the number of elements being processed is not an exact multiple of 64, the resulting "remainder" is processed first.

It is important to realize all types of data used in scalar registers, i.e.,floating-point, integer, logical, and even address data, can be used in vector registers. Furthermore, the functional units of the vector portion of the machine enable us to use any of these types. However, the CFT compiler, because of the great complexity involved, sometimes does not efficiently use the full capabilities of the hardware for non floating-point data.

A special feature of vector supercomputers such as the CRAY-X/MP is the ability to chain 2 or more vector operations. By this we mean the results coming out of one vector operation may feed directly as operands into another vector operation without waiting for the first to complete. Not only computational units, but also vector memory references can chain also. The basic flow of chaining is shown is figure 1.3 for a SAXPY operation (whose signifficance will be discussed in the Linear Algebra section).

```
SAXPY       DO 100 I=1,M
            Y(I) = Y(I) + A(I)*X
100         CONTINUE
```

2.0 MEMORY. First we examine some details of the CRAY X-MP vector-memory system in order to discuss applications later.

2.1. ORGANIZATION. The 2^{23} (=8,388,608), 64-bit words of central memory for a CRAY X-MP are arranged into 64 banks, each of 2^{17} words. These 64 banks are in turn grouped into four sections. Each processor or CPU has its own set of four lines to memory, one to each of the four sections. Thus the CRAY X-MP/48 has a total of 16 section lines (4 CPUs with four lines each). The CPUs access the lines through three ports, A and B for vector reads, and C for vector stores (there is also a fourth I/O port). This is summarized in figure 2.1.

2.2. VECTOR MEMORY REFERENCES The hardware instructions for vector memory references (reads and stores) each come in three modes. Mode one is for adjacent memory locations, i.e. for stride one references, where stride is the constant number of words from one vector element to the next consecutive one. Mode two is the generalization to stride n references. Thus for stride three store, we mean we store our data in memory in every third word from some starting location. The third mode is a generalization of the second mode to irregular locations in memory. This is called gather for read and scatter for store. These hardware instructions require a second vector register with the relative addresses in memory from which to gather or to which to scatter. We could, in FORTRAN, gather into vector Y irregularly located items of vector A (using index array K) and add them to scalar X by

```
      DO 100 I = 1, N
      Y(I) = X + A(K(I))
100   CONTINUE
```

We could do the analogous scatter by

```
      DO 100 I = 1, N
      Y(K(I)) = X + A(I)
100   CONTINUE
```

Vector memory references generate bona fide vector streams that perform up to 64 operations for each instruction. These instructions can be chained (linked, overlapped vector streams, as shown below) with vector arithmetic units. Furthermore, with bidirectional memory enabled, the two vector reads (ports A and B) can be chained with a vector store (port C). Thus the X-MP is ideally designed for the SAXPY discussed in the previous chapter.

2.3. ACCESS CONFLICTS. With four CPUs using 12 ports and 16 section lines to reference central memory, there are clearly possibilities for access conflicts! The three basic types of memory access conflicts are Bank Busy, Simultaneous Bank, and Section Access. The Bank Busy conflict occurs within or between CPUs when a port requests a bank that is already busy. Referring to figure 2.1, suppose port B of CPU 2 is already reading from bank 4 in section 1. If either port A in CPU 2 or port C in CPU 3 wishes to access that same bank, then we will have a bank busy conflict. Since a bank is closed for a total of four cycles upon being requested (the initial request cycle plus three more cycles), a Bank Busy conflict will result in a delay of one, two, or three cycles to the new requesting port. A Simultaneous Bank conflict results when ports of different CPUs make an initial bank request at the same time. This would occur, for example, if port B in CPU 0 and port C in CPU 2 both wanted bank 17 at the same time. All ports are held one cycle by a Simultaneous Bank conflict and a Bank Busy conflict will immediately follow for a least one of the ports. A Section Access conflict occurs when multiple ports in the same CPU request banks in the same section. An example of this would be port C in CPU 3 requesting bank 28 and port A in CPU 3 requesting bank 12, since both banks 12 and 28 are in section 3. All ports are held one cycle by Section Access conflicts. The access conflicts are resolved as follows:

i. Within a CPU, ports with odd stride (including stride one) have precedence over even stride requests and prior requests have priority over later requests.
ii. Between CPUs, the priority between CPUs is rotated every four cycles.

2.4. VECTOR STRIDE. Next we examine vector stride, where stride is the number of words from one array element to the next. This is of particular interest because there are separate hardware memory reference instructions to implement unit stride (stride steps of one) and nonunit stride (steps greater than one).

We next need to examine the FORTRAN methods of generating memory array references. To show this, we consider the one-dimensional array, W, the two-dimensional array X, the three-dimensional array Y, and the five-dimensional array Z, defined by

```
DIMENSION  W(21),  X(4,5),  Y(3,2,4),  Z(4,2,3,4,10)
```

The elements of W are stored in their natural order, i.e., W(1), W(2), W(3), . . . , W(21). For the multi-dimensional arrays X, Y, and Z, the elements are stored in reverse lexicographic fashion, i.e., leftward indices vary more rapidly than rightward ones. More generally, for an n dimensional array, U, defined in FORTRAN by

$$\text{DIMENSION}\ \ U(D_1,D_2,\ldots,D_n)$$

The relative position, $\mathcal{P}$, of a particular element, $U(\delta_1,\delta_2,\ldots,\delta_n)$, is given by

$$\mathcal{P}(\delta_1,\delta_2,\ldots,\delta_N) = \delta_1 + \sum_{J=1}^{N-1} \{(\delta_{j+1} - 1) * (\prod_{k=1}^{j-1} D_k)\} \qquad (2.1)$$

Thus Z(3,1,2,3,7) is the 635th element of Z since $\mathcal{P}(3,1,2,3,7) = 3 + (2-1)*4*2 + (3-1)*4*2*3 + (7-1)*4*2*3*4 = 635$. Unit stride in FORTRAN is illustrated below for both the arrays W (reads) and Y (stores).

```
      DO 100 I = 1, 10
      DO 100 J = 1, 4
      TOTAL = TOTAL + W(I+3)
      Y(J,2,1) = COS(J*PI)
100   CONTINUE
```

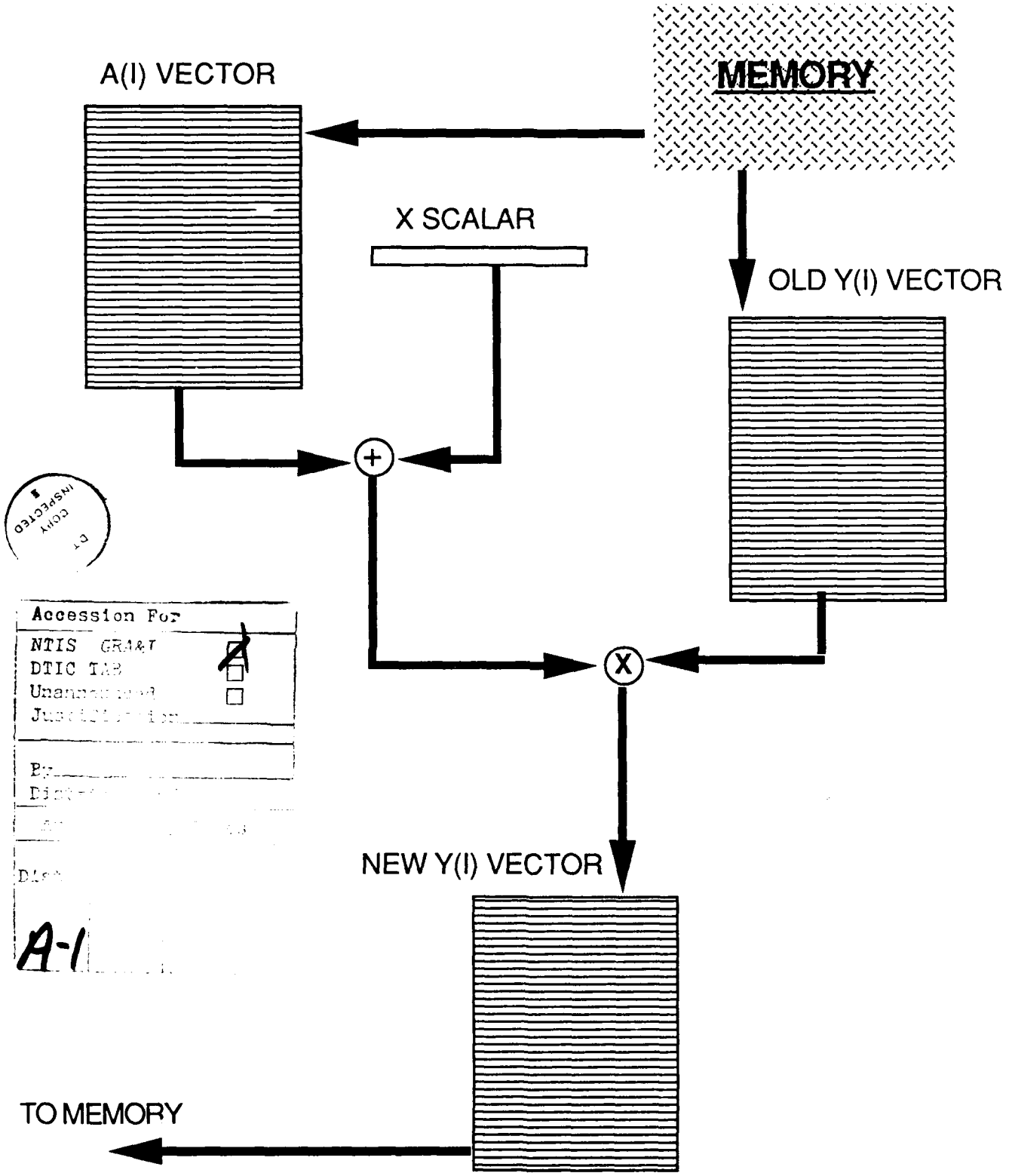


Figure 1.3. *SAXPY chaining flow, Y(I) = Y(I) + A(I)*X.*

By contrast nonunit stride in FORTRAN is shown below for stride-of-five reads for W, and stride-of-eight stores for Z.

```
      DO 100 I = 1, 20, 5
      DO 100 J = 1, 3
      TOTAL = TOTAL + W(I)
      Z(3,1,J,4,7) = SIN(J*PI)
100   CONTINUE
```

2.4.1. STRIDE CONFLICTS. The performance and memory conflict characteristics of stride can be divided into three basic categories: unit stride, nonunit odd stride, and nonunit even stride. Here we discuss the simplest and generally best behaved case, unit stride. The low-order six bits of the absolute memory address specify one of the 64 banks. Therefore unit stride will access successive banks and will cycle sequentially through all banks before returning to any starting bank. Since there are no more than 64 elements in a vector register, a unit stride vector reference in a 64 bank CRAY will not access any bank more than once, and hence will never have a bank conflict with itself. In the case of section line conflicts, unit stride (and odd stride) have priority over even stride. Furthermore, once unit stride takes a line, it will hold it for up to four cycles, thus reducing certain cases (linked conflicts) in which a second port of a CPU continually contends with a first for line access. In all these senses, unit stride avoids or reduces conflicts of the kind often seen for both odd and even nonunit stride.

Now we consider the behavior of different types of stride. Nonunit odd stride behaves basically like unit stride, primarily because odd stride also cycles through all banks. To see this, we observe the number of banks, 64, is a power of two. Hence any odd number, s, for the stride will be relatively prime to the number of banks. From elementary number theory, this means the set of bank numbers accessed by unit stride, $\{k\}_{0\le k\le 63}$, will be the same as those accessed by odd stride, s, i.e., $\{ks \pmod{64}\}_{0\le k\le 63}$. Thus, nonunit odd stride, like unit stride, will cycle through all banks (and section lines) before returning to a starting bank (or line), though the actual sequence order of accessed banks will be different than for unit stride. The general conclusion (both within and between CPUs) is *Memory conflict characteristics for unit stride and nonunit odd stride are similar*.

2.4.1.1. STRIDE CONFLICT EXAMPLE. This is shown by an experiment for which MFLOPS speeds (averaged over 100,000 iterations) were computed for stride increments, s = 1, 2, . . . , 64 for the code

```
      DO 100 I=1,(64*INCR),INCR
      C(I) = (T * A(I)) + B(I)
100   CONTINUE
```

The analogous result with two gathers and a scatter was also computed.

```
      DO 100 I=1,64
      C(INDEX1(I)) = (T * A(INDEX2(I))) + B(INDEX3(I))
100   CONTINUE
```

Define $\deg_k(n)$ = highest power of k in factorization of n, e.g., $\deg_2(7) = 0$, $\deg_2(12) = 2$, $\deg_2(58) = 1$, and $\deg_3(54) = 3$. Next define $\Delta_j = \{ n : \deg_2(n) = j, 1 \le n \le 64\}$, so that Δ_0 is the odd intergers {1, 3, 5, . . ., 63}, Δ_3 = {8, 24, 40, 56}, Δ_5 = {32}, and Δ_6 = {64}. Define Δ_∞ to refer to the gather/scatter case. Fig. 2.2 shows the results of the experiment. In fact, if we define $\mathfrak{M}_n$ to be the average megaflops in the experiment above correpsonding to stride classes in Δ_n, then we found the $\mathfrak{M}$'s to be strictly ordered, i.e., $114.8 = \mathfrak{M}_0 > \mathfrak{M}_1 > \ldots > \mathfrak{M}_6 = 28.2$. Thus the results (figure 2.2) show

i. Unit stride and odd nonunit stride give about the same results and these are the best cases.
ii. Even stride gives poorer performance than odd stride and increasing "degree" of evenness decreases the performance
iii. Gather/scatter compares with some degree of evenness.

Although exceptions can be found, these results are quite general.

One important reason for this phenomenon is even stride does not cycle through all the banks as odd stride does. Since the number of banks is 64, the case of Δ_1 will access only half the banks. In the case of Δ_2 only a quarter, etc. This means the bank conflicts due to random access from other processors will be magnified because of reduced possibilities, i.e., free banks, to avoid conflicts. In addition, cases Δ_3 through Δ_6 will not cycle through all section lines and hence cause an analogous problem for a processor with itself (if it is doing multiple memory references, as in our example). Also in the cases of Δ_4 through Δ_6 we expect our processor to have bank conflicts with itself since the bank busy period is four cycles. Finally we note that Δ_∞ results are markedly inferior to Δ_0, since Δ_∞ produces random memory accesses which will tend to conflict with itself. However these are not as bad as Δ_5 or Δ_6 since in those latter cases we are always guaranteed long bank busy conflicts.

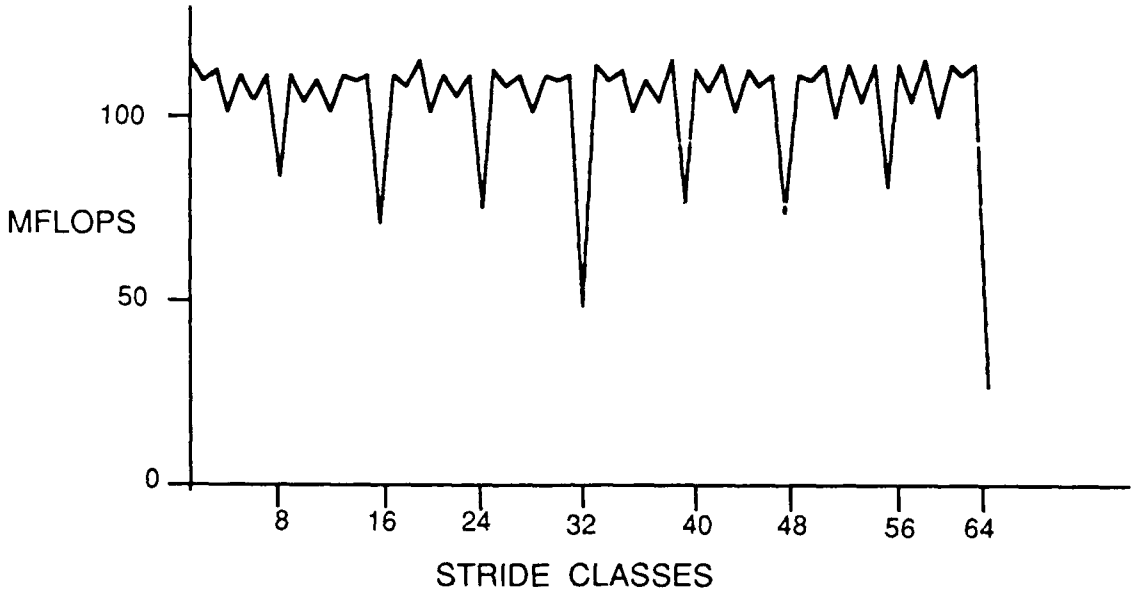


Figure 2.2. *Stride contention effects on vectorizing of* $c_i = t*a_i + b_i$.

3.0. LINEAR ALGEBRA.

3.1. MATRIX TIMES A VECTOR. Let us examine chaining in a basic linear algebra operation, matrix times a vector. Let $X^T = (X(1),X(2), \ldots, X(N))$ be a point in N-space and $A = (A(I,J))$ be the M by N matrix mapping X into M-space.

$$\begin{pmatrix} A(1,1), & A(1,2), & \ldots, & A(1,1) \\ A(2,1), & A(2,12, & \ldots, & A(1,1) \\ & \ldots, & & \\ & \ldots, & & \\ A(M,1), & A(M,2), & \ldots, & A(1,1) \end{pmatrix} \begin{pmatrix} X(1) \\ X(2) \\ \\ \\ X(N) \end{pmatrix} = \begin{pmatrix} Y(1) \\ Y(2) \\ \\ \\ Y(M) \end{pmatrix} \qquad (3.1)$$

where $Y(I) = \sum_{J=1}^{N} A(I,J) * X(J)$, I=1,. . . M.

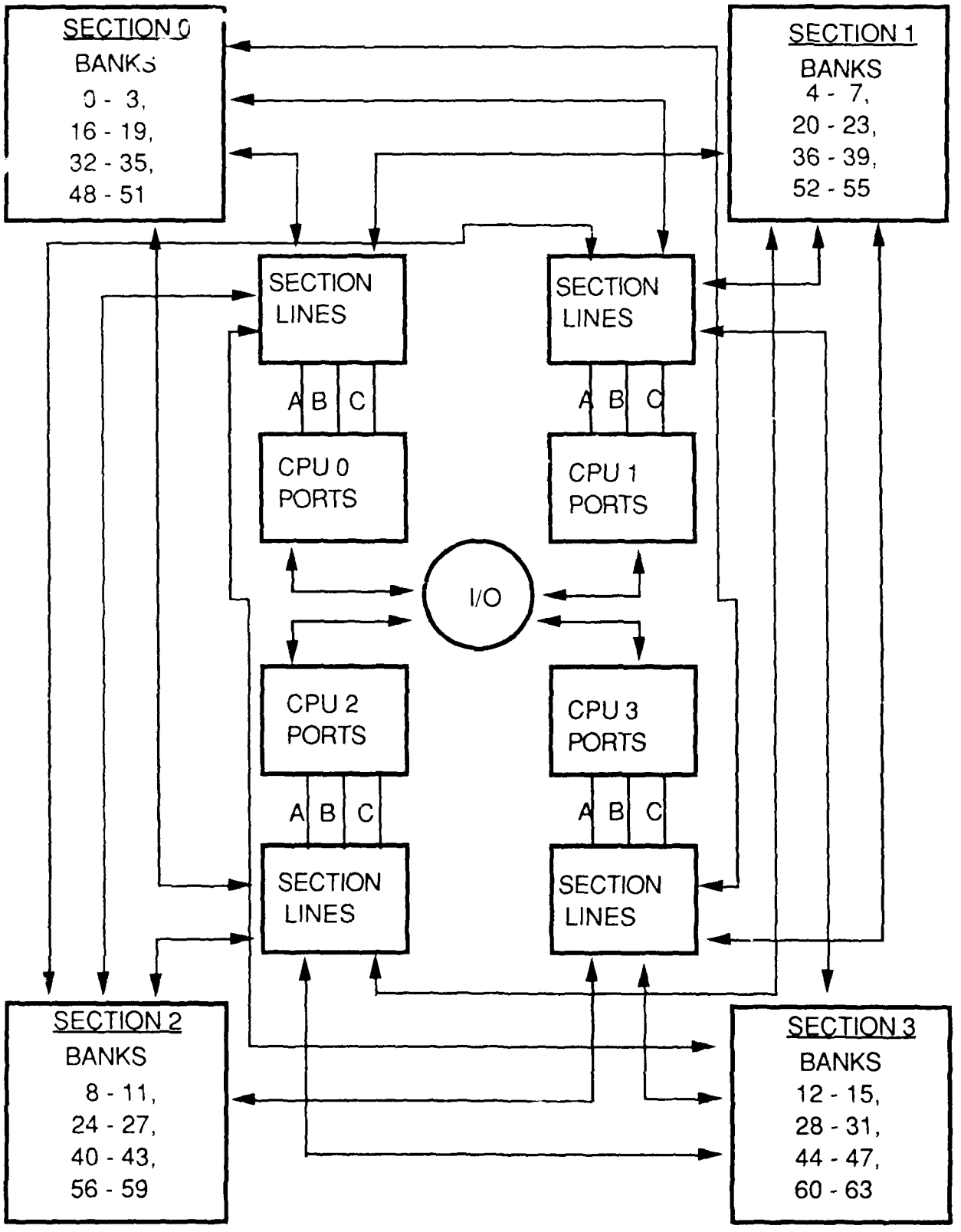


Figure 2.1. *CRAY X-MP central memory*

3.2. SAXPY AND SDOT. Algorithmically we can think of this in two ways: (a) as N updates to the elements of **Y** by successively adding in terms A(I,J)*X(J), commonly called SAXPY, or (b) as M dot products of rows of **A** with the column vector of **X** aka SDOT. The FORTRAN for both is below (assuming initialization of **Y** to 0).

```
SAXPY       DO 100 J=1,N
            DO 100 I=1,M
              Y(I) = Y(I) + A(I,J)*X(J)
100         CONTINUE
```

```
SDOT        DO 100 I=1,M
            DO 100 J=1,N
              Y(I) = Y(I) + A(I,J)*X(J)
100         CONTINUE
```

Figures 3.1 and 3.2 show timings for the inner loop chains of SAXPY and SDOT. In particular they show the high degree of concurrent multiplies and adds for both algorithms. In addition vector loads and stores are chained with arithmetic operations. Since each chain can only operate on up to 64 elements (the maximum vector length), these chains may be repeated, depending on the lengths of M or N. Note neither the load of X(J) for SAXPY nor the store of Y(I) for SDOT are shown since these are both one-time scalar operations with respect to the iterated inner loops. On a CRAY X-MP, SAXPY is faster for small and medium data sizes (M≤750) and SDOT for large data sizes (N≥750). Using information about the CRAY's memory organization and vector hardware (discussed above) we can deduce the reasons for this. Dongarra et. al. [1984] have explained this only on the basis of memory touches and possible bank conflicts. These are actually, however, usually only secondary factors. Since in the way we have written SDOT above, the inner loop is not on the leftmost FORTRAN variable, then we have nonunit stride through memory. Section 2 (Memory) above showed that certain (highly even) strides can be quite bad. However, most strides, odd or only slightly even, do not cause much degradation in performance. In any case, the results are still the same: SAXPY is better for small and medium data; SDOT for large data.

3.3 PERFORMANCE ANALYSIS. By examining figures 3.1 and 3.2, we see the SDOT chime (chain time) is about 10 percent shorter than the SAXPY chime, since it does not have the vector store operation that SAXPY uses. How then can SAXPY ever take less time? The reason lies in the one-time, 64-long, scalar summation that has to be done at the end of the SDOT. This stems from the difficulty of vector machines to perform reduction operations, i.e., collapse the contents of a vector to a scalar value. What we want to compute in the last line of SDOT above is a vector reduction of the generic type, $\text{scalar}_1 = \text{scalar}_1 + (\mathbf{vector}_1 \otimes \mathbf{vector}_2)$ where $\otimes$ denotes component-wise multiplication. However the result register of a vector operation must generally be another vector register. Thus we must use a calculation of the generic type $\mathbf{vector}_0 = \mathbf{vector}_0 + (\mathbf{vector}_1 \otimes \mathbf{vector}_2)$. This means $\mathbf{vector}_0$ will hold Y(I), but split up into 64 partial summands, i.e., the μth word of $\mathbf{vector}_0$ contains

$$Y_\mu(I) = \sum_{J \equiv \mu (\mathrm{mod}\ 64)} A(I,J) * X(J) \qquad (3.2)$$

What we want to compute is

$$Y(I) = \sum_{J=1}^{N} A(I,J) * X(J) = \sum_{\mu=0}^{63} Y_\mu(I) \qquad (3.3)$$

To do this we must perform a scalar sum of the words of $\mathbf{vector}_0$ at the end. For relatively small vectors the duration of this one-time add can be large compared to the basic SDOT work, in which case the overall SAXPY performance will be faster. However since this scalar SDOT summation is only one-time, its effect will be relatively insignificant for larger vectors and SDOT will then be the faster.

4.0. CONCLUSION. Memory contention can have severe effects on the perfomance of vector machines. These effects can be minimized by insuring that memory references have as small a power of 2 as possible in the stride size. SDOT is asymptotically more efficient than SAXPY, however the inability of vector machines to fully implement SDOT makes SAXPY better for small and medium lengths.

REFERENCES

CRAY Y-MP System Programmer Reference Manual, CRAY Research Inc., 1988.

J. J. Dongarra, F.G. Gustafson, A. Karp, *Implementing Linear Algebra Algorithms for Dense Matrices on a Vector Pipeline Machine*, SIAM Review, vol. 26, No. 1, January 1984.

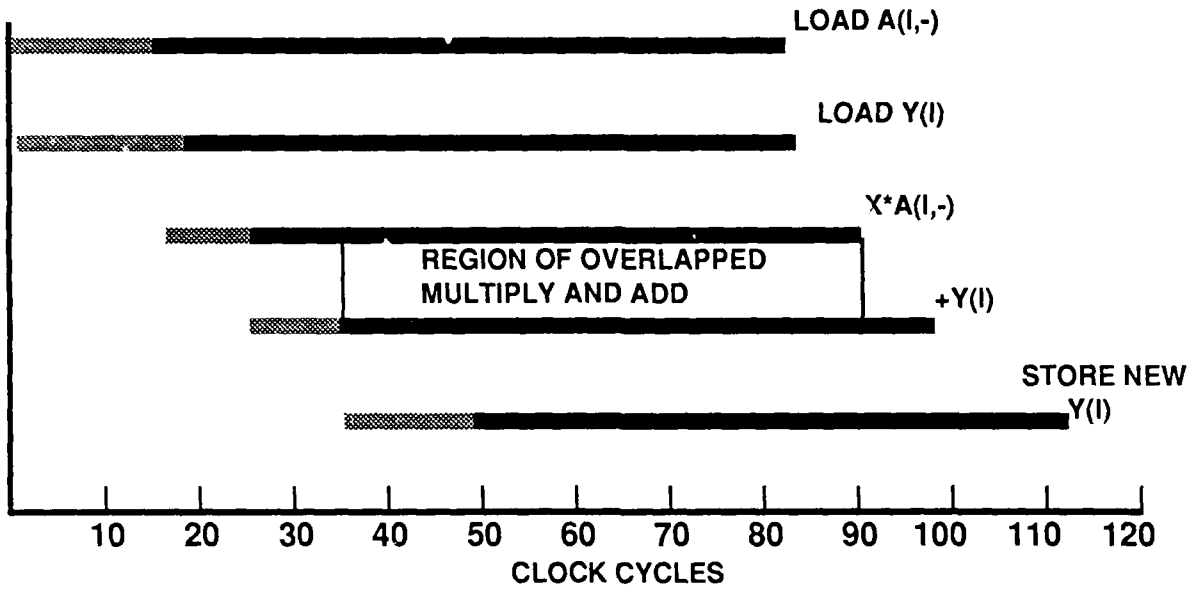
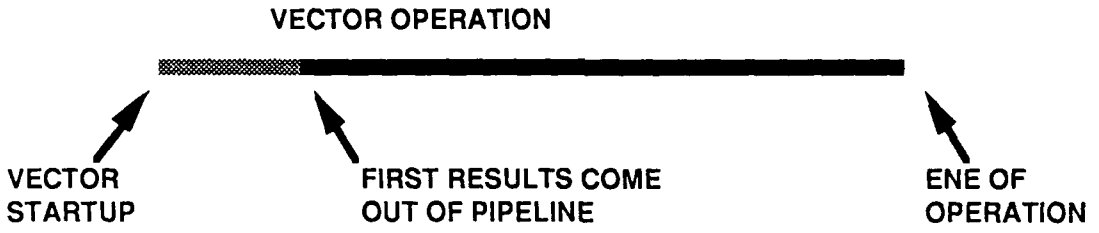


Figure. 3.1 *SAXPY*



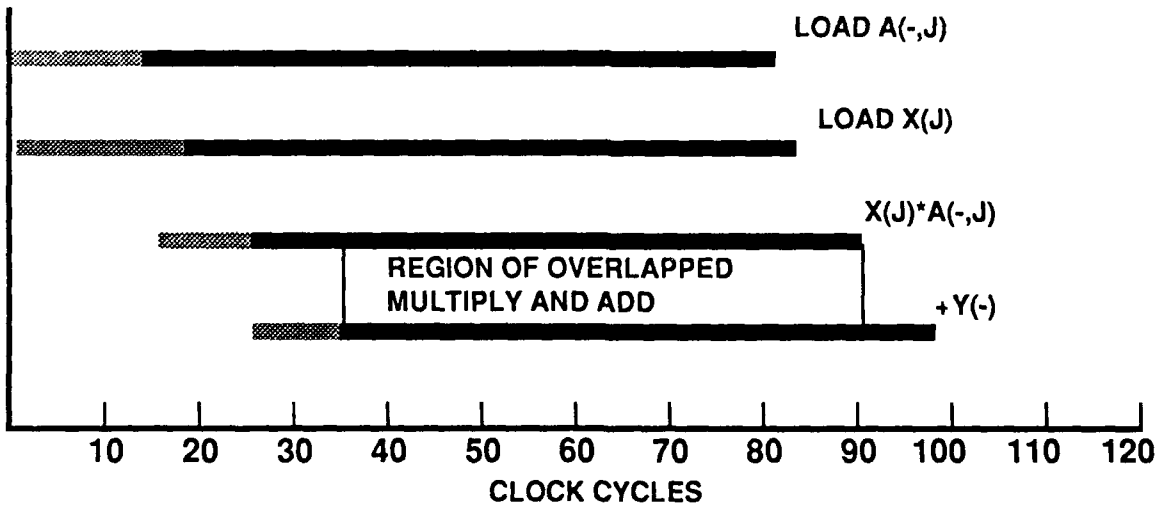


Figure 3.2. *SDOT*

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	April 1990	Final

4. TITLE AND SUBTITLE

LINEAR ALGEBRA ON A CRAY X-MP

5. FUNDING NUMBERS

PE: 060223
WU: DN306243

6. AUTHOR(S)

R. F. Freund

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Naval Ocean Systems Center
San Diego, CA 92152-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

NOSC TD 1670

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Naval Ocean Systems Center
Block Programs
San Diego, CA 92152-5000

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This paper discusses basic issues of vectorization as well as memory organization and contention for vector machines. There is an analysis of the implications of these issues for the performance of basic linear algebra operations, SAXPY and SDOT.

14. SUBJECT TERMS

computer architecture
multiprocessor systems
operating systems

15. NUMBER OF PAGES

15

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	SAME AS REPORT